Providing Hints, Next Steps and Feedback in a Tutoring System for Structural Induction

Providing Hints, Next Steps and Feedback in a Tutoring System for Structural Induction

Josje Lodder
Faculty of Management, Science and Technology
Open University of the Netherlands
Heerlen, The Netherlands
josje.lodder@ou.nl

Bastiaan Heeren
Faculty of Management, Science and Technology
Open University of the Netherlands
Heerlen, The Netherlands
bastiaan.heeren@ou.nl

Johan Jeuring
Department of Information and Computing Sciences,
Universiteit Utrecht, The Netherlands
j.t.jeuring@uu.nl

Structural induction is a proof technique that is widely used to prove statements about discrete structures. Students find it hard to construct inductive proofs, and when learning to construct such proofs, receiving feedback is important. In this paper we discuss the design of a tutoring system, LogInd, that helps students with constructing stepwise inductive proofs by providing hints, next steps and feedback. As far as we know, this is the first tutoring system for structural induction with this functionality. We explain how we use a strategy to construct proofs for a restricted class of problems. This strategy can also be used to complete partial student solutions, and hence to provide hints or next steps. We use constraints to provide feedback. A pilot evaluation with a small group of students shows that LogInd indeed can give hints and next steps in almost all cases.

1 Introduction

Discrete structures play an important role in many domains, and are foundational for mathematics, logic, and computer science. Examples of such structures are natural numbers, data structures such as lists and trees, but also complex structures such as programming languages. Structural induction is a proof technique that is widely used to prove statements about inductively defined, discrete structures. Mathematical induction can be viewed as a special kind of structural induction, using the inductive definition of the natural numbers as the underlying structure.

Because discrete structures and structural induction are foundational for mathematics and computer science, they form an integral part of educational programs. For example, proof techniques are part of the ACM Computer Science curriculum.[1] Courses that address proof techniques often require students not only to learn how to prove consequences in a formal system, but also to reason about formal systems, and to independently construct a proof for a statement. A typical example of an exercise that occurs in many textbooks on logic and proof techniques is the following: prove that the number of left parentheses in a logical formula is equal to the number of right parentheses. Such a property can be proved by structural induction, where the structure of the proof follows the structure of the inductive definition of the logical language. Students have to learn this proof technique to construct more fundamental proofs, such as the soundness of a proof system. Textbooks and teachers typically instruct students on how to do this, provide some examples, and then let students practice with constructing proofs themselves. As

[1] https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf

P. Quaresma, W. Neuper and J. Marcos (Eds.): Theorem proving components for Educational software 2019 (ThEdu'19)
EPTCS 313, 2020, pp. 17–34, doi:10.4204/EPTCS.313.2

with learning any subject, students need feedback when they are learning how to construct their own proofs [14]. Such feedback can take several forms: it may be about the progress of a student, about recommending a next task to solve, or about the difference between the proof constructed by a student and an expected proof. In this paper we focus on the latter kind of feedback.

This paper discusses the design of a tutoring system for practicing proving statements about inductively defined, discrete structures. Some core features of the system are that it gives feedback on the steps a student takes towards a solution, and hints about which step to take next. We use the knowledge about misconceptions students have to determine what feedback to give. Students have some freedom in setting up their proof, and the system helps in reaching the learning goal of independently constructing a proof for statements about inductively defined discrete structures. Two advantages of our system are that it gives immediate feedback, and that it is scalable because feedback is calculated automatically.

This paper is organized as follows. After introducing our terminology in Section 2, we discuss related work in Section 3. Our research is partly motivated by students' problems with induction, discussed in Section 4. Section 5 describes the interface and functionality of LogInd, and in Section 6 we show how this functionality is realized. A pilot experiment is discussed in Section 8, and we conclude in Section 9 with conclusions and ideas for future work.

2 Terminology

We use an example exercise to illustrate the terminology concerning inductive proofs that we use in this paper. The text of the exercise is:

"The propositional language L has atoms $p, q, r, ...$ and connectives $\neg$, $\wedge$ and $\rightarrow$. We define two functions on this language: a function *prop* counting all occurrences of propositional letters, and a function *bin* counting the number of binary connectives. These functions are inductively defined by:

$$\begin{array}{ll} prop\ (p) & = 1 \\ prop\ (\neg\phi) & = prop\ (\phi) \\ prop\ (\phi \square \psi) & = prop\ (\phi) + prop\ (\psi) \end{array}$$

$$\begin{array}{ll} bin\ (p) & = 0 \\ bin\ (\neg\phi) & = bin\ (\phi) \\ bin\ (\phi \square \psi) & = bin\ (\phi) + bin\ (\psi) + 1 \end{array}$$

where p is an atom, and $\square$ is $\wedge$ or $\rightarrow$. Prove with induction that $prop\ (\phi) = bin\ (\phi) + 1$ for any formula ϕ in the language L."

The statement $prop\ (\phi) = bin\ (\phi) + 1$ in the last sentence is the *theorem* or *property* that has to be proven. The structure of an inductive proof for this theorem can be deduced from the inductive definition of the language L. The *base case* consists of a proof of the theorem for atomic formulae. There is an *inductive case* for each of the connectives in the language. For instance, a proof of the conjunction case is a proof that from the assumption that if the theorem holds for ϕ and ψ (i.e. $prop\ (\phi) = bin\ (\phi) + 1$ and $prop\ (\psi) = bin\ (\psi) + 1$) it follows that the theorem also holds for $\phi \wedge \psi$ (i.e. $prop\ (\phi \wedge \psi) = bin\ (\phi \wedge \psi) + 1$). The assumption that the theorem holds for some arbitrary formulae ϕ and ψ is the *induction hypothesis*. A *subproof* is a part of the complete inductive proof where a base case or inductive case is proven.

3 Related work

Tools that assist a user in constructing structural induction proofs may have different functionalities or purposes. In this section we describe four different kinds of tools: automated theorem provers, proof assistants with didactic functionality, e-learning tools for mathematical induction, and e-learning tools for structural induction.

Bundy [7] states that Gödel's incompleteness theorem implies that it is impossible to construct a completely automatic inductive theorem prover, and that Kreisel's result on cut-elimination [17] implies that inductive proofs in general will need intermediate lemmas. Hence, automated theorem provers for induction are proof assistants that use several heuristics to try to automatically prove theorems as much as possible. The classical literature on automated theorem proving, for example the handbook of Boyer-Moore [5], describes strategies to find structural induction proofs. This includes different ways of using the induction hypothesis (weak and strong fertilization), selection of the induction variable, and the recognition of the need for extra lemmas. More recent research adds the use of rippling as an important technique [6]. To use an automated theorem prover a user should have a thorough understanding of inductive proofs. Because the concept of induction is the learning goal in courses on proof techniques, using such provers in education is in general not very helpful.

Proof assistants such as Tutch [1] and Minifn [24] have been developed for educational purposes. Tutch is based on a high-level proving language, which allows step sizes resembling the steps in a pen-and-paper proof. Minifn tries to integrate functional programming with mathematical induction in a way that students can quickly learn how to use the proof assistant. Although these proof assistants are much easier to use than regular theorem provers, they remain assistants. They do not offer exercises, nor do they provide feedback or hints.

A couple of e-learning tools support learning mathematical induction. In EAsy, an e-assessment system, a student practices with different kinds of proof exercises, using rules and proof strategies from a drop-down menu [12]. For a problem requiring a proof by induction, the system pre-structures the proof in a base case and an inductive case, and it provides the induction hypothesis. The system performs a selected rule, which ensures that a student cannot make a mistake in applying a rule. Since there are quite a lot of rules and the exercises are non-trivial, a student easily can get lost: in an evaluation 41% of the students mentions having problems in selecting the right rule from the extensive ruleset. Completed proofs are graded automatically, but incomplete proofs have to be graded by hand. EAsy shows a student that she completed a proof or subproof successfully, but does not provide any other feedback.

In the Intelligent Book, a student practices exercises on mathematical induction [4]. This e-learning system uses MathTiles, predefined templates, which have to be completed by the student. An automated theorem prover, Isabelle, is used in the backend to check correctness and provide simplifications. As in EAsy, the system automatically generates the goals for the different cases. The authors state that this is necessary since Isabelle only accepts these goals when they are exactly equal to the goals in the prover. Simplification can be performed automatically, but when, for example, an application of the induction hypothesis is needed, simplification is not accepted. The tool provides hints in some cases, based on teacher scripts.

ComIn-M contains electronic exercise sheets [28]. The system is developed as part of the SAiL-M project, and extends an earlier e-assessment tool [22]. These sheets also contain pre-structured mathematical induction exercises, and combine multiple choice exercises about stating the induction hypothesis with open exercises to prove a base case and an inductive case. Students have to state the inductive case before proving it. A nice feature of ComIn-M is the possibility to work in two directions. Feedback and hints are provided automatically.

The website[2] accompanying the textbook Discrete Mathematics: Mathematical Reasoning and Proof with Puzzles, Patterns and Games by D. Ensley and W. Crawley [9] contains a set of interactive exercises in which given a property P, a student first has to deduce $P(n+1)$ from $P(n)$ for concrete values, and then for an arbitrary value n. In this way, a student gets some intuition behind induction before she proves the inductive case.

We found only two systems addressing structural induction. The most extensive system is Polycarpou's e-book, which is a complete educational environment for learning structural induction. She emphasizes foundational concepts, such as structures, sets and closed sets. A separate chapter introduces inductive definitions. Animations show how these definitions generate inductive sets. Interactivity is restricted to multiple choice, drag and drop, and fill in the blank exercises, which implies that a student does not independently complete an inductive proof.

Stanford's online Introduction to Logic course[3] does offer the possibility to construct complete structural induction proofs in its open online logic course. However, in the course material induction is combined with natural deduction, causing long and abstract proofs. It is possible to ask for a complete solution to an exercise, but the system does not give feedback on mistakes, nor hints on how to proceed. The site also offers a proof assistant that combines structural induction with Hilbert-style proofs.

4 Students' problems with structural induction

Students have problems with constructing inductive proofs. Several studies identify misconceptions students have with mathematical induction, and analyze the underlying reasons for these misconceptions [3, 8, 10, 13, 25, 26]. In her thesis, Polycarpou studies possible causes of problems students have with structural induction [27]. Her hypothesis is that a lack of understanding of set theoretical concepts is one of the main causes of these problems. To investigate this hypothesis, she performed an experiment in which students have to answer six questions about a fancy inductively defined language IPO (a fictive programming language). The base elements of this language are lower case characters. There are two inductive rules to construct new words: by concatenating two IPO words by an underscore, and by putting quotes around an IPO word. The first four questions test whether a student understands this definition: the student has to indicate which words are IPO words in a given set of words, give the minimal length of an IPO word, determine whether or not IPO words have a maximum length, and construct an IPO word of length greater than 6. The fifth question prepares for the last question by asking if it is possible to construct a word of length 8. Finally, in the last question students have to tell whether an IPO word can have even length, and in case the answer is no, prove that all IPO words have odd length. Students participating in this experiment are enrolled in a course 'Logic for Computer Science' and they have practiced with mathematical induction in an earlier course on discrete mathematics. As a pre test, students have to answer these exercises two months before the lessons on induction. After these (traditional classroom) lessons a comparable set of exercises is given as a post test.

In the pre test, students particularly experience problems with the exercise on identifying IPO words, the problem about a word with length 8, and the inductive proof. The results on these questions are much better in the post test, but still the inductive proof is too hard for 44% of the students. The author claims that there is a correlation between performance on the first five questions and performance on the inductive proof, but this correlation is not statistically motivated. Instead, she calculates the ratio between students who perform well on both the inductive set exercises and the inductive proof exercise

[2] http://higheredbcs.wiley.com/legacy/college/ensley/0471476021/anim_flash/index.html
[3] http://intrologic.stanford.edu/public/index.php

(71%), and the ratio between students who do not perform well on both (83%). She notes that some students find an (incorrect) IPO word with length 8, but manage to prove that all IPO words have odd lengths. Another observation is that some students seem to copy an example treated in the course without the necessary adaptations. These students define inductive cases for negation and conjunction, instead of for the IPO constructors. The conclusion of Polycarpou is that lack of understanding of an inductive definition is indeed a main cause of problems with induction, and that the traditional way of teaching results in procedural knowledge without conceptual understanding for some students.

To investigate whether students entering the master Computer Science at the Open University of the Netherlands, a distance learning university, experience similar problems, we analyze the solutions to a homework assignment of 20 of these students. Before being admitted to the Computer Science master program, students have to take a course in logic. Structural induction is one of the topics in this course. Unlike the students participating in Polycarpou's experiment, almost none of these students has experience with mathematical induction. Furthermore, their background in mathematics is usually weaker than that of bachelor students at a regular university. The homework assignment is optional, but gives extra credits at the exam.

In the first part of the homework assignment (Exercise 1), students have to give an inductive definition of a function *len*, which returns the length of a propositional formula. The next question (Exercise 2) asks to give an inductive definition of a postfix function $*$ that rewrites all conjunctive subformulae $\phi \wedge \psi$ of a formula into the equivalent subformula $\neg(\neg\phi \vee \neg\psi)$. The last question (Exercise 3) asks for an inductive proof of the following property: $len\,(\phi*) \leqslant 3\,len\,(\phi) - 2$ for all formulae ϕ. This assignment differs from Polycarpou's: it does not test the understanding of inductively defined sets, but instead tests the understanding of inductively defined functions (in exercises 1 and 2).

As Polycarpou, we expect correlations between performance on the first two exercises and the last exercise. Since the number of participating students is too low for a statistic test, we perform a similar calculation as Polycarpou. Students who do not receive full points for the first two exercises (11 students), are almost all (10) unable to complete the third exercise. However, from the students who receive full points (9) only 2 manage to complete the inductive proof. We conclude that for these students Exercise 3 is too difficult to complete without help, but most students have fewer problems with the inductive function definitions. Table 1 shows the results on the first two exercises. Most students provide a correct definition, but some students add an induction hypothesis to this definition. Since students can make several mistakes, for example, 'no correct cases' and 'an incorrectly added induction hypothesis', the sum of the percentages in Table 1 (and also Table 2) is more than 100%.

We looked further into mistakes students made in the proof exercise. As shown in Table 2, the inductive part of the proof most often goes wrong, and the most common error is assigning a fixed length to a compound formula (for example, $len\,(\phi \wedge \psi) = 3$). The inductive definition does not seem the bottleneck, since 70% of the students (see the columns 'correct' + 'correct use of IH, but incomplete' + 'correct ind def, no use of IH' in Table 2) apply these definitions in their proof. The assumption $\phi* = \phi$ made by some students could be a symptom of conceptual misunderstanding of an inductive definition, but could also be used because a student does not know how to apply the induction hypothesis. We conclude that although problems with inductive definitions may certainly play a role in the results in the inductive proofs, the understanding of the role of the induction hypothesis and the way this hypothesis can be used is perhaps a more important cause of problems. We think that the remedy of Polycarpou (an intelligent tutoring system that mainly focuses on theoretical foundations) probably will not be the best solution for our students, since the concept of inductive sets does not seem to be their main problem, and her approach might be too theoretical. Instead we concentrate on an e-learning tool that guides students interactively through the construction of an inductive proof.

Table 1: Results for exercises 1 and 2 of the homework assignment

(a) Exercise 1

solution	(N = 20)
correct base and ind. cases	80%
only correct base case	5%
one missing case	5%
no correct cases	10%
incorrectly added IH	15%

(b) Exercise 2

solution	(N = 20)
correct	60%
no base case	30%
incorrect	10%

Table 2: Results and mistakes for Exercise 3 of the homework assignment (N = 20)

solution	base	IH	induction
correct	75%	50%	15%
IH only for one formula ϕ	–	20%	–
assumption ϕ and ψ atomic	–	–	35%
assumption $\phi* = \phi$	–	–	15%
correct use of IH, but incomplete	–	–	10%
correct ind def, no use of IH	–	–	45%
IH as goal	–	–	10%
verbal intuitive argument	–	–	5%

5 LogInd, a tool for teaching structural induction

This section describes LogInd, a tool that supports students with constructing inductive proofs. Experience with other intelligent tutoring systems for logic (LogEx for rewriting propositional formulae [19], and LogAx for Hilbert-style axiomatic proofs [18]) shows that students benefit from a system where they can enter solutions stepwise, get feedback after each step, and can ask for a hint or next step at any moment, or receive a worked-out solution. The possibility to add proof steps both backwards and forwards in these systems resembles the way an exercise is solved with pen and paper. We use the same approach in LogInd. Since some students have hardly any idea how to start an inductive proof, LogInd offers guidance in structuring the proof. We also want students to be aware of the kind of steps they perform in the proof, for example, applying the induction hypothesis or an inductive definition of a given function. Hence, LogInd asks for a justification at each step. Students do not have to justify simple calculation steps, such as distributing a multiplication over an addition, and we allow calculation steps at different levels of granularity. Therefore, LogInd checks calculations by normalizing the submitted expression.

LogInd guides a student through a proof by structuring the proof in three parts: a proof of the base case, stating the induction hypotheses, and a proof of the inductive cases. After presenting the exercise, LogInd asks the student first to state what is to be proven in the base case, see Figure 1. If this is correct the student is asked to complete the proof of the base case, and to continue with stating the induction hypotheses, see Figure 2. For the inductive cases, LogInd again first asks what the different cases are, and what has to be proven in these cases. A complete proof is shown in Figure 3.

LogInd uses a domain reasoner to provide hints, next steps, feedback, and complete solutions. A domain reasoner is an expert module that performs all reasoning about the domain [11]. Thus far, LogInd only offers exercises about properties of a propositional language. We introduce the term ‘counting

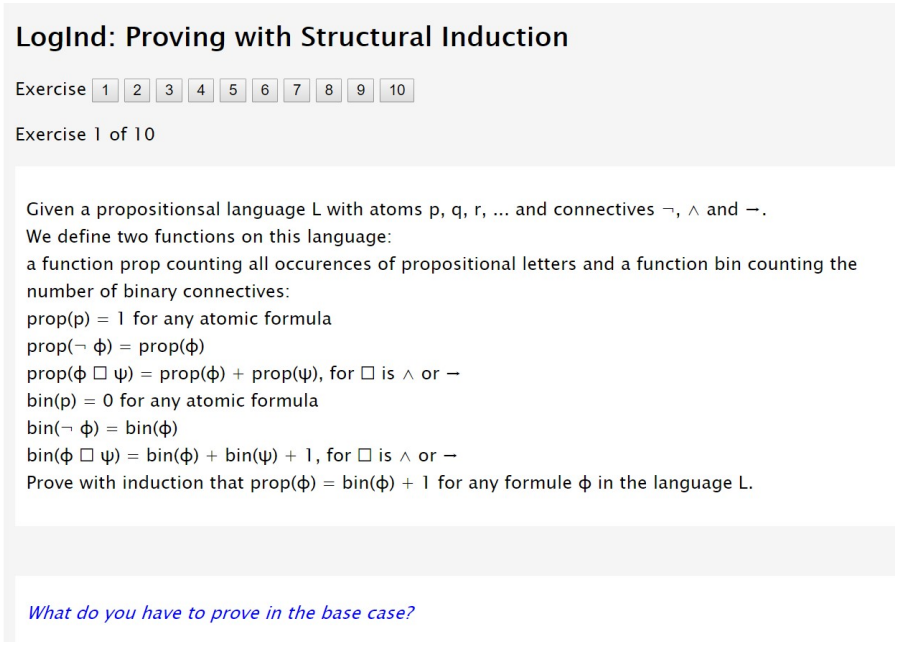

Figure 1: Starting the first exercise in LogInd

function' to describe the set of exercises that can be used in LogInd.

A counting function *count* is an inductively defined function such that

$$\begin{array}{lll} count\ (p_i) & = c_i, & c_i \in \mathbb{N}, \\ count\ (\neg\phi) & = a + b \cdot count\ (\phi), & a, b, \in \mathbb{N} \\ count\ (\phi\ \square\ \psi) & = a_\square + b_\square \cdot count\ (\phi) + c_\square \cdot count\ (\psi), & a_\square, b_\square, c_\square \in \mathbb{N} \end{array}$$

where p_i is a propositional letter, and $\square$ a binary connective.

The properties that have to be proven take the following form: $P_1\ (\phi)\ comp\ P_2\ (\phi)$, where *comp* is a comparator $(=, <, \leqslant, >, \geqslant)$ and $P_i\ (\phi), i = 1, 2$ is either a truth value, number or formula:

- if $P_i\ (\phi)$ is a truth value, it is an expression $V\ (g\ (\phi))$ or a constant where V is a valuation and g an inductive function from the language L to L;
- if $P_i\ (\phi)$ is a natural number, the right-hand side is a linear combination of expressions $f\ (g\ (\phi))$ where f is a counting function and g an inductively defined function from L to L, the left-hand side is a single expression $f\ (g\ (\phi))$;
- if $P_i\ (\phi)$ is a a formula, it is equal to an expression $g\ (\phi)$ where g is an inductively defined function from L to L;
- valuations V may have predefined properties such as $V\ (p) = 1$, but if the comparator in the theorem is an equality, these properties may also only make use of equalities (since in our proof system it is not possible to prove an equality from inequalities).

The restriction in the second option that the left-hand side is a single term $f\ (g\ (\phi))$ while the right-hand side may contain a linear combination of terms is not a real restriction, since a statement where both the left- and right-hand side contain linear expressions can always be rewritten into this form. The restriction will enable us to treat the induction hypothesis as a rewrite rule. Examples of exercises in this format are:

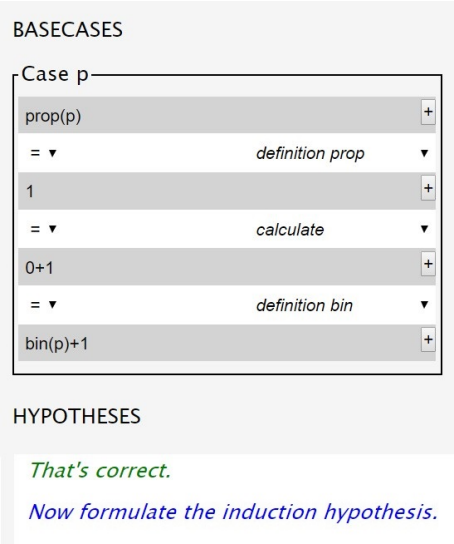

Figure 2: Guidance after the base case is finished

- Let L be a propositional language with connectives $\wedge$ and $\vee$. Let *ValA* and *ValB* be two valuations such that $\mathit{ValA}(p) \leqslant \mathit{ValB}(p)$ for any atomic formula p. Then $\mathit{ValA}(\phi) \leqslant \mathit{ValB}(\phi)$ for any formula ϕ in the language L.
- Let L be a propositional language with connectives $\wedge$ and $\vee$, and L' the extension of L with negation $\neg$. The function *star* from L to L' replaces every atom by its negation, conjunctions by disjunctions and disjunctions by conjunctions. The function *length* returns the length of a formula. The following holds: $\mathit{length}(\mathit{star}(\phi)) \leqslant 2 \cdot \mathit{length}(\phi)$
- Let L be a propositional language with connectives $\neg$, $\wedge$, $\vee$ and $\rightarrow$. Function f replaces every conjunctive subformula $\phi \wedge \psi$ by $\neg(\neg\phi \vee \neg\psi)$, and function g replaces every implicative subformula $\phi \rightarrow \psi$ by $\neg\phi \vee \psi$. Then $f(g(\phi)) = g(f(\phi))$ (where '=' means syntactically equal) for any formula ϕ.

This class of problems offers sufficient possibilities for relatively simple exercises, where students can get acquainted with inductive proofs. In the next section we show that for this class we can generate solutions, hints and next steps, without the need for advanced techniques as used by automatic theorem provers.

6 Generation of solutions, hints and next steps

In general, automatic proof generation for induction problems is undecidable [2, 7]. Problems that might seem easy, such as the proof for associativity of list concatenation for a single list $((l :: l) :: l = (l :: l) :: l)$, already need advanced methods to be proven automatically (in this case generalization of the first occurrence of l) [7]. Automatic theorem provers use sophisticated techniques such as rippling and lemma generation to solve inductive problems [6, 7]. It is not our goal to teach students these techniques, and by restricting ourselves to the class of problems described in the previous section, we only need a straightforward strategy to solve such exercises.

The problem-solving strategy we use is part of our domain reasoner. The strategy first uses the definition of the language to decide what has to be proven in the base cases, and which inductive cases

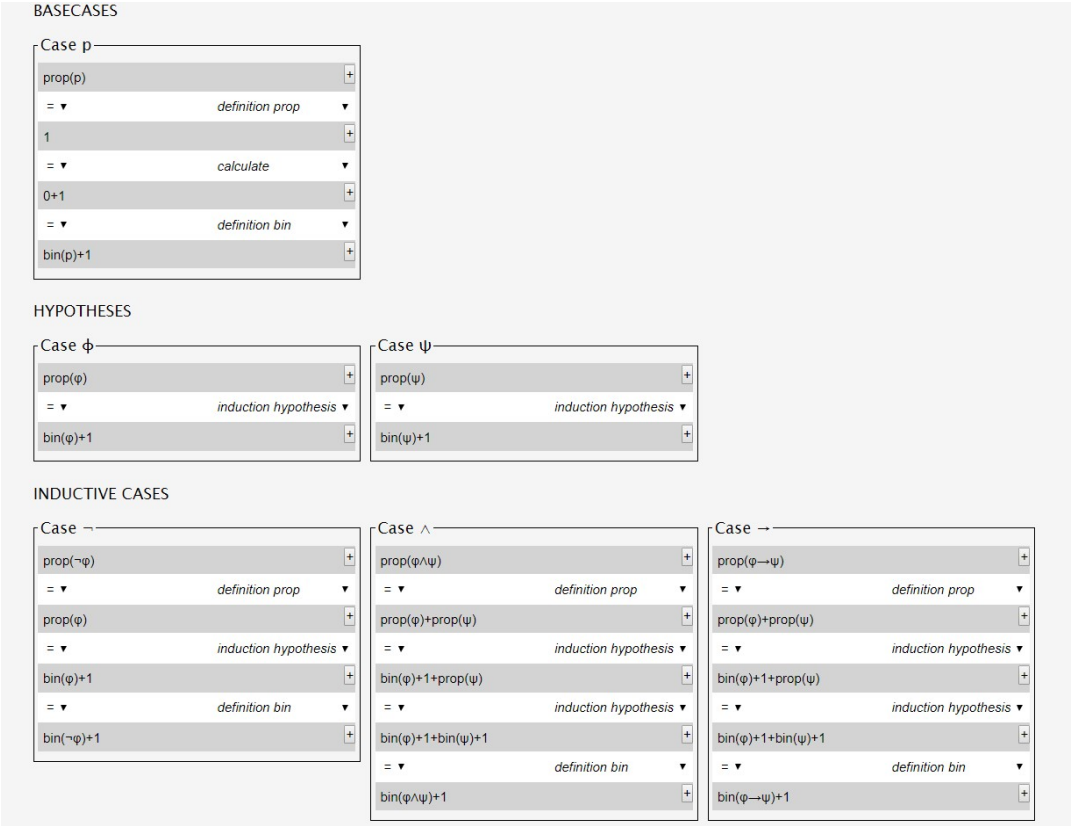

Figure 3: The solution of the exercise of Figure 1 in LogInd

have to be treated. We only allow a single formula variable in a property, so we do not have to ask which variable will be used for induction. Inductive functions are represented as rewrite rules, just as the induction hypothesis. Apart from some technical details, the strategy first applies the inductive definitions of the functions occurring in the statement. The strategy rewrites both the right-hand and left-hand side of the statement. Hence, the strategy supports the possibility to complete a subproof by working in two directions. In case the inductive function has natural numbers as the codomain, the next step (if necessary) is a distribution of the multiplication such that the left-hand side and right-hand side become linear combinations of terms that occur in the induction hypothesis. Now we can apply the induction hypothesis to occurrences of the left-hand side of this hypothesis in the left-hand side formula in the proof. After this application only some normalizing elementary calculations might be needed to complete the proof. For our restricted class of problems this strategy always finds a solution. We provide a sketch of a proof of this statement in Appendix A. Students will not always follow this strategy. For example, they might apply the induction hypothesis before rewriting the right-hand side of a statement applying the inductive definitions, or vary in the calculations. Also in these cases, LogInd continues with applying the strategy. Hence, a next step can be provided as long as the strategy's rules are applicable. After application of these rules, normalization is sufficient to complete the proof. We expect that almost all student steps will be such that LogInd can indeed provide a hint or next step based on the student solution. The evaluation described in Section 8 gives evidence for this claim.

Our strategy differs from the method advocated by Bundy [7]. The difference is in the way we use the induction hypothesis. Bundy recommends strong fertilization: the isolation of the induction hypothesis

in an inductive case and replacement of this hypothesis by True. Our strategy uses weak fertilization: substituting the left-hand side of the induction hypothesis by the right-hand side. Bundy advises the use of strong fertilization since the use of weak fertilization generally results in longer and more complicated proofs. For our restricted class of problems, this is not the case. Since most pen-and-paper proofs use weak fertilization, we also use weak fertilization in our strategy.

7 Constraints and feedback

From the analysis of the homework assignment, we expect our students to make various kinds of mistakes while practicing with LogInd. Examples of potential mistakes are:

- treating metavariables ϕ and ψ as atoms, for example, resulting in the rewriting of *length* $(\phi \wedge \psi)$ into 3 in the proof of an inductive case;
- omission of a case, for example negation;
- use of only $=$ and $\leqslant$ when a statement $P_1(\phi) < P_2(\phi)$ has to be proven;
- forgetting to state the induction hypothesis before using it.

In our other tutoring systems for logic we use buggy rules to generate feedback in case a student makes a mistake. Buggy rules typically relate to mistakes on the level of single steps. An example of such a buggy rule is forgetting to change a disjunction in a conjunction in an application of DeMorgan while rewriting a formula into normal form. Mistakes in an inductive proof can be on the level of a step, for instance rewriting *length* $(\phi \wedge \psi)$ to 3, but also on the level of a subproof. An example of an error in a subproof is using $\leqslant$ between each of the steps when the goal is to prove an equality. In this case, each of the steps is correct, but the overall relation between the first and last line of the proof is $\leqslant$ instead of $=$. An example of an error on the level of the whole inductive proof is the omission of a case. These mistakes are easily formulated as constraint violations. For example, the composition of the relations between the lines in a proof should imply the relation between left-hand side and right-hand side in the theorem that is proven. We think that constraints can be put to good use for this domain.

Ohlsson [23] first described the role of constraints in learning. Mitrovic used the concept in the development of an SQL tutor [20] and many other tutors. Constraints characterize correct solutions by providing a relevance condition and a satisfaction condition: if the relevance condition holds, the solution should satisfy the satisfaction condition. Some important reasons for using constraints in the development of tutoring systems are that constraints partially play the role of buggy rules, the construction of which is very time consuming, and that constraints can also be used to give feedback if student solutions diverge from model solutions or are partial [20, 21].

LogInd uses constraints to provide feedback and guidance. Heeren and Jeuring [15] describe the diagnose service used by a domain reasoner to provide feedback. Figure 4 is based on this description, and shows how we incorporate constraints in the diagnosis. Our diagnose service receives a (partial) student solution and checks whether or not this submission violates a set of constraints. We divide the constraints in constraints on the level of steps, on the level of subproofs, and on the level of proofs.

- Constraints on the level of steps check:
  - if the rewriting of a line (for example the application of an inductive definition) is correct;
  - if the induction hypothesis is applied correctly;
  - if comparators (=, $<$ ..) are used correctly in a single step;
  - if the justification is correct.

- Constraints on the level of subproofs check:
  - if the first and last line of each subproof are instances of the left-hand side respectively right-hand side of the theorem;
  - if these instantiations are valid cases (atomic base cases, inductive cases only for connectives in the language);
  - if the induction hypothesis (when present) is correctly formulated;
  - if comparators (=, $<$, ..) are used correctly at the level of subproofs (i.e. if the composition of the comparators in the subproof equals the comparator in the theorem).
- Constraints on the level of complete proofs check:
  - if all inductive cases correspond to connectives in the language;
  - if the induction hypothesis is stated before use, with the same metavariables;
  - if an inductive case or special base case is missing.

To check a proof at the level of a step, the domain reasoner compares the student submissions with the result of the application of possible rules, and accepts the student submission if it is similar to one of these results. Here, a simple normalizing calculation transforms the submission and the generated result into the same expression. For example, application of the induction hypotheses in the example of Figure 3 results in $bin\ (\phi) + 1 + bin\ (\psi) + 1$, but a submission $bin\ (\phi) + bin\ (\psi) + 2$ is also accepted.

If no constraint is violated, the domain reasoner compares the new submission with the previous (last correct) submission, and determines if these are similar. If these submissions are similar, the domain reasoner gives feedback about this. A submission that is not similar may follow the implemented strategy, which is diagnosed as 'expected'. When the step does not follow the strategy, but is recognized by the domain reasoner, the diagnosis is a 'detour'. This happens, for example, when a student starts with completing the inductive case for implication before negation. The interface will tell the student that this step is correct, and the student can continue with the exercise. Since there are no violations, every step in the submission is already recognized, which means that the last option (no rule detected) only happens when the student submission contains more than one new line. Again, the interface will provide a message 'this is correct'.

Our experience is that the diagnosis 'failure' of a constraint is not enough to provide informative feedback. For example, a constraint on the exercise in Figure 1 could be that the first line of an inductive case should be an instantiation of the left-hand side of the theorem $prop\ (\phi) = bin\ (\phi) + 1$, with ϕ substituted by $\neg\alpha$, $\alpha \wedge \beta$ or $\alpha \rightarrow \beta$, where $\alpha \neq \beta$ and $\alpha, \beta \in \{\phi, \psi\}$. Now a student can violate this constraint in different ways, for example by

- using a connective that is not in the language;
- instantiating with an atomic formula;
- instantiating with a metavariable that is not used in the induction hypothesis;
- introducing an expression that is not an instantiation at all.

Each of these violations stem from a different misconception, and we want to give different feedback messages in each case. We solve this by specifying failure messages for different constraints, and call this use of constraints 'buggy constraints' as proposed by Kodaganallur et al. [16]. One of the problems of the use of constraints as mentioned by Kodagallur et al. [16] and Mitrovic [20] is the violation of two or more constraints at the same time. We solve this by ordering the constraints: for example, constraints about instantiations get a higher priority than constraints about the application of a rule.

Apart from 'strong' constraints that may not be violated, we also use soft constraints to guide a

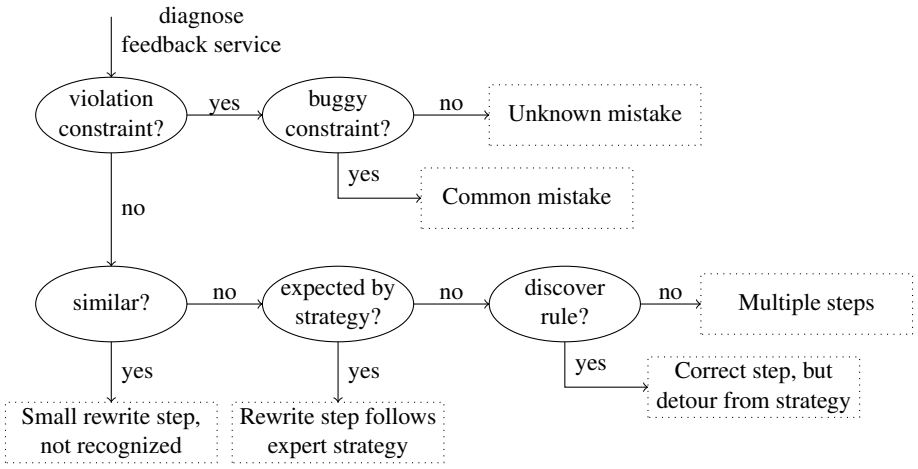

Figure 4: Structure of the diagnose feedback service: the incorporation of constraints is new

student through a proof. These constraints check for each of the subproofs if they are introduced and if they are finished. After a diagnosis, the user interface can call the feedback service 'constraints', which reports the status of each of the subproofs. The result can be used to provide a message such as: 'the base case is finished, continue with the formulation of the induction hypothesis'.

The way we use constraints in LogInd differs in some aspects from the original use. One difference is the fact that LogInd has a strategy which produces solutions, and while checking the next step of a student, LogInd first tries to recognize this step. LogInd can hence be conceived as a constraint-based solver as described in Kodaganallur et al. [16]. Moreover, we use constraints not only to provide feedback on errors, but also to guide a student.

8 Evaluation

In April 2019 we performed a small pilot experiment with a group of 15 students taking an online logic course in preparation of admission to the master in Computer Science at the Open University of the Netherlands. Before the experiment, these students handed in the homework assignment described in Section 4. The experiment consisted of an online instruction about the use of LogInd, followed by the possibility to practice. During the experiment students could ask questions by using the chat functionality of the learning environment. The questions that we would like to answer with this experiment are:

1. does LogInd behave as expected, i.e., provide hints and next steps, and give a correct diagnosis?
2. what kind of problems do students have while working with LogInd?
3. how do students use LogInd?
4. can we see effects of the use of LogInd in the way students perform pen-and-paper exercises?

In Section 6 we claim that we can provide a hint or next step at any moment, also if a student does not follow the preferred strategy by LogInd. We analyze the loggings to check this claim. From the 1612 calls obtained from student interactions (a diagnosis, a hint, a next step, or a full solution), 398 calls ask for a hint or a next step. In 25 cases (6%) LogInd cannot provide such a step. Further analysis shows that since students repeat their call several times, this only happens in five different cases (1%). In all

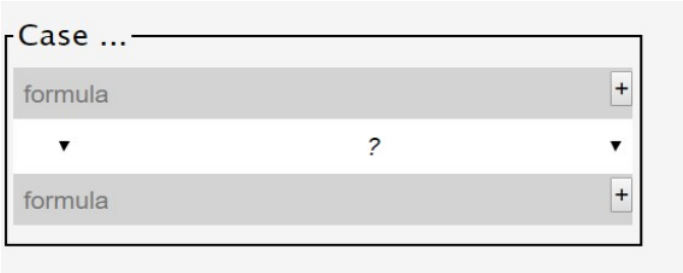

Figure 5: Template for starting a base case

of these cases LogInd could not provide a hint or next step because it diagnoses the exercise as 'ready'. This was a bug, resulting from a use of LogInd that we had not anticipated: some students did not start a base case with the statement that they have to prove, but they submit the first two lines of a proof of the base case, for example the subproof:

$$\begin{array}{l} \mathit{prop}\ (p) \\ = (\mathit{definition}\ \mathit{prop}) \\ 1 \end{array}$$

In such a case LogInd decided that this step was correct and that this subproof was finished since all steps are motivated, without checking whether indeed the base case has been proven. When a student asked for a hint after the other subproofs were (correctly) finished, LogInd could not provide such a hint. We repaired this bug, and since this was the only reason that no hint or next step was available, we expect that LogInd now indeed provides this kind of feed forward (i.e. hints and next steps) in all circumstances.

We use the remarks made by students in the chat during the experiment and the loggings to answer the second question. From these remarks and the loggings we learned that students had quite a lot of problems with the interface. Students should start an exercise with a response to the question 'what do you have to prove in the base case?'. They should fill in their answer in a template as shown in Figure 5. For the first exercise, the exercise of the example in Section 2, this means that on the first line, a student should enter $\mathit{prop}\ (p)$, then choose the equality sign (=) from the drop-down list and enter the right-hand side $\mathit{bin}\ (p)+1$ in the bottom line. In their first attempt, none of the participating students entered this first step correctly. Ten students did not realize that they first had to answer this question before completing the proof or did not know what to prove. They provided answers such as for example $\mathit{prop}\ (p) = 1$ or $\mathit{prop}\ (\phi) = \mathit{bin}\ (\phi)+1$. Three students entered the whole statement $\mathit{prop}\ (p) = \mathit{bin}\ (p)+1$ on the first line, one student added a wrong justification (definition of the inductively defined function *prop*), and one student replaced the ? by an empty string, which at that time was not accepted by LogInd.

A second source of problems was the use of the send button. A student only gets feedback after clicking this button. If a student enters several lines before clicking this button, it was hard to find the place where the feedback referred to, especially since at the time of the experiment we were still developing the constraints. So a student might receive the message 'this step is not correct' without a clue which step should be repaired. Also, when a student asked for a hint after receiving an error message, this hint was based on the latest correct submission. Hence, this hint might relate to the application of a rule in (for example) a base case, while the student was working on an inductive case.

To answer the third question, how do students use LogInd, we also looked at the loggings. The problems described in the previous section combined with the conceptual and technical problems students have with induction caused a high hint and next step use (25% of the student interactions in the

Solution	LogInd (N = 7)		No LogInd (N = 4)	
	1st	2nd	1st	2nd
Correct	–	2	–	–
Assumption ϕ and ψ atomic	3	1	4	1
Assumption $\phi* = \phi$	1	3	0	1
Correct use of IH but incomplete	1	–	–	–
Incorrect use of IH	–	1	–	–
Correct ind def, no use of IH	3	5	4	2

Table 3: Results and mistakes in the first submission of homework assignment 3 and in the improved second submission (number of students)

loggings). However, students did enter steps themselves and asked for a diagnosis (1078 requests, 67%), where 619 steps were diagnosed as correct. Half of the participating students were able to construct (parts of) a subproof, some of them using hints or next steps in between, but the other students had too many problems with the interface. Students who solved the homework assignment before the experiment, could use LogInd without too many problems.

To answer the last question, we analyzed resubmissions of students' homework. Students whose homework assignment was not correct had to submit the assignment again. We hoped to use these improved assignments as a kind of post test. However, three students submitted their corrections before the experiment, and one student did not submit a new version. The results of the remaining seven students can be found in Table 3. We use the same characterizations of solutions as in Table 2, except for the solution label 'incorrect use of IH', which did not occur in the first submissions. The first column shows the results on the first submissions of the homework assignment by the group of students who practiced with LogInd and submitted a new version after the experiment. The results of this second submission is shown in the second column. The last two columns show the same data for the group of students who did not practice with LogInd. The second submission is in both groups better than the first attempt. The number of students is too low to conclude if practicing with LogInd has more effect than the comments by the teacher on the first attempt.

9 Conclusion and future work

We have discussed the design of a tutoring system for learning how to prove statements about inductively defined discrete structures. As far as we know, LogInd is the first such tutoring system. A student constructs her proof stepwise, and LogInd provides help (hints, next steps, and elaborate feedback on errors) at each step. A pilot evaluation showed that LogInd indeed can provide help in almost all situations. Half of the students in the experiment could construct (parts of) a proof by themselves, but the other half had too many problems with the interface and the exercises. The number of students who participated in the experiment was too low to decide whether students indeed learn by using LogInd. We noticed that homework submissions by students who had practiced with LogInd were more clearly structured, and contained more justifications of different steps.

In a next experiment we will evaluate the constraints: is the information in a feedback message correct, and do these messages help students to correct their submission? We will test an alternative interface, where students can enter their steps in a text field, and we will also compare a guided version

with a non-guided version. In the future we might also incorporate exercises about induction in other domains, such as for example lists or functional programs.

Acknowledgments

We thank Aad van Lieburg and Anja Paalvast for their work on the student interface, and our students for their permission to use their solutions in our research, and their participation in the experiment.

References

[1] Andreas Abel, Bor-Yuh Evan Chang & Frank Pfenning (2001): *Human-Readable Machine-Verifiable Proofs for Teaching Constructive Logic*. In Uwe Egly, Armin Fiedler, Helmut Horack & Stephan Schmitt, editors: *Proceedings of the Workshop on Proof Transformations, Proof Presentations, and Complexity of Proofs, Siena 2001*, pp. 37–50, doi:10.1.1.25.4339.

[2] Raymond Aubin (1979): *Mechanizing structural induction part I: Formal system*. *Theoretical Computer Science* 9(3), pp. 329 – 345, doi:10.1016/0304-3975(79)90034-3.

[3] Shmuel Avital & Shlomo Libeskind (1978): *Mathematical induction in the classroom: Didactical and mathematical issues*. *Educational Studies in Mathematics*, pp. 429–438, doi:10.1007/BF00410588.

[4] William Billingsley & Peter Robinson (2007): *Student Proof Exercises Using MathsTiles and Isabelle/HOL in an Intelligent Book*. *Journal of Automated Reasoning* 39(2), pp. 181–218, doi:10.1007/s10817-007-9072-3.

[5] R.S. Boyer & J.S. Moore (1998): *A Computational Logic Handbook*. Academic Press International series in Formal Methods, Academic Press, doi:10.1016/C2013-0-10412-6.

[6] A. Bundy (2005): *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge Tracts in Theoretica, Cambridge University Press, doi:10.1017/CBO9780511543326.

[7] Alan Bundy (2001): *The Automation of Proof by Mathematical Induction*. In: *Handbook of Automated Reasoning (in 2 volumes)*, pp. 845–911, doi:10.1016/b978-044450813-3/50015-1.

[8] Ed Dubinsky (1986): *Teaching mathematical induction 1*. *The journal of mathematical behavior*, 5(3), pp. 305–317.

[9] Douglas E. Ensley & J Winston Crawley (2006): *Discrete Mathematics with Student Solutions Manual Set*. Wiley.

[10] Paul Ernest (1984): *Mathematical induction: A pedagogical discussion*. *Educational Studies in Mathematics* 15(2), pp. 173–189, doi:10.1007/BF00305895.

[11] George Goguadze (2010): *ActiveMath - generation and reuse of interactive exercises using domain reasoners and automated tutorial strategies*. Ph.D. thesis, doi:10.22028/D291-26097.

[12] S. Gruttmann, D. Böhm & H. Kuchen (2008): *E-Assessment of Mathematical Proofs — Chances and Challenges for Students and Tutors*. In: *Proceedings of the 2008 International Conference on Information Technology in Education*, Wuhan,China. doi:10.1109/CSSE.2008.95.

[13] Guershon Harel (2001): *The development of mathematical induction as a proof scheme: A model for DNR-based instruction*. In R. Zaksis S. Campbell, editor: *Learning and teaching number theory*, Kluwer Academic, pp. 185–212, doi:10.1023/A:1023682216523.

[14] John Hattie & Helen Timperley (2007): *The power of feedback*. *Review of Educational Research* 77(1), pp. 81–112, doi:10.3102/00346543029848.

[15] Bastiaan Heeren & Johan Jeuring (2014): *Feedback services for stepwise exercises*. *Science of Computer Programming, Special Issue on Software Development Concerns in the e-Learning Domain* 88, pp. 110–129, doi:10.1016/j.scico.2014.02.021.

[16] Viswanathan Kodaganallur, Rob R. Weitz & David Rosenthal (2005): *A Comparison of Model-Tracing and Constraint-Based Intelligent Tutoring Paradigms*. *Int. J. Artif. Intell. Ed.* 15(2), pp. 117–144. Available at `http://dl.acm.org/citation.cfm?id=1434925.1434928`.

[17] Georg Kreisel (1965): *Mathematical logic*. In T Saaty, editor: *Lectures on modern mathematics*, 3, John Wiley & Sons, Inc., New York, London, and Sydney, pp. 95–195.

[18] Josje Lodder, Bastiaan Heeren & Johan Jeuring (2017): *Generating Hints and Feedback for Hilbert-style Axiomatic Proofs*. In Michael E. Caspersen, Stephen H. Edwards, Tiffany Barnes & Daniel D. Garcia, editors: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, WA, USA, March 8-11, 2017*, ACM, pp. 387–392, doi:10.1145/3017680.3017736.

[19] Josje Lodder, Bastiaan Heeren & Johan Jeuring (2019): *A comparison of elaborated and restricted feedback in LogEx, a tool for teaching rewriting logical formulae*. *Journal of Computer Assisted Learning* 35(5), pp. 620–632, doi:10.1111/jcal.12365.

[20] Antonija Mitrovic (2012): *Fifteen years of constraint-based tutors: what we have achieved and where we are going*. *User Modeling and User-Adapted Interaction* 22(1), pp. 39–72, doi:10.1007/s11257-011-9105-9.

[21] Antonija Mitrovic, Brent Martin & Pramuditha Suraweera (2007): *Intelligent Tutors for All: The Constraint-Based Approach*. *Intelligent Systems, IEEE* 22, pp. 38–45, doi:10.1109/MIS.2007.74.

[22] Wolfgang Müller & Maren Hiob-Viertler (2010): *Intelligent Assessment in Math Education for Complete Induction Problems*. In Xiaopeng Zhang, Shaochun Zhong, Zhigeng Pan, Kevin Wong & Ruwei Yun, editors: *Entertainment for Education. Digital Techniques and Systems: 5th International Conference on E-learning and Games, Edutainment 2010, Changchun, China, August 16-18, 2010. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 317–325, doi:10.1007/978-3-642-14533-9.

[23] Stellan Ohlsson (1994): *Constraint-Based Student Modeling*. In Jim E. Greer & Gordon I. McCalla, editors: *Student Modelling: The Key to Individualized Knowledge-Based Instruction*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 167–189, doi:10.1007/978-3-662-03037-0.

[24] Peter-Michael Osera & Steve Zdancewic (2013): *Teaching Induction with Functional Programming and a Proof Assistant*. In: *SPLASH Educators Symposium (SPLASH-E), 2013*.

[25] Marina Palla, Despina Potari & Panagiotis Spyrou (2012): *Secondary school students understanding of mathematical induction: structural characteristics and the process of proof construction*. *International Journal of Science and Mathematics Education* 10(5), pp. 1023–1045, doi:10.1007/s10763-011-9311-2.

[26] Margrital Pavlekovič (1998): *An approach to mathematical induction - starting from the early stages of teaching mathematics*. *Mathematical communcations*, pp. 135–142. Available at `https://hrcak.srce.hr/1798`.

[27] Irene Polycarpou (2008): *An Innovative Approach to Teaching Structural Induction for Computer Science*. Ph.D. thesis, Florida International University. doi:10.25148/etd.FI08121912.

[28] Sandra Rebholz & Marc Zimmermann (2013): *Applying Computer-Aided Intelligent Assessment in the Context of Mathematical Induction*. In Zhigeng Pan, Adrian David Cheok, Wolfgang Müller, Ido Iurgel, Paolo Petta & Bodo Urban, editors: *Transactions on Edutainment X*, *Lecture Notes in Computer Science* 7775, Springer Berlin Heidelberg, pp. 191–201, doi:10.1007/978-3-642-37919-2_11.

A Sketch of a completeness proof for the strategy used by LogInd

We give a sketch of the completeness proof for the strategy used by LogInd. First we specify the class of functions that are used in in our exercises for transforming a formula in another formula. We call these inductively defined functions acceptable.

Definition 1. *An inductively defined function g from the propositional language L to L is acceptable if the inductive definition can be written in the following form:*

- $g\,(p_i) \quad = \phi_i \quad$ *for atomic formula* p_i
- $g\,(\neg\phi) \quad = [g\,(\phi)\,/\,s]\;\psi$
- $g\,(\phi_1 \,\square\, \phi_2) = [g\,(\phi_1)\,/\,s_1, g\,(\phi_2)\,/\,s_2]\;\psi_\square$

The definition states that the inductive cases are obtained by substituting a variable s by $g\,(\phi)$ in a formula ψ, or the variables s_1 and s_2 by $g\,(\phi_1)$ and $g\,(\phi_2)$ in a formula $\psi_\square$. All inductive definitions of functions from L to L can be written this way. For example, the function *star* in the second example in Section 5 can be defined by:

$$\begin{aligned}
g\,(p_i) &= \neg\, p_i \\
g\,(\neg\phi) &= [g\,(\phi)\,/\,s]\,(\neg s) \\
g\,(\phi_1 \wedge \phi_2) &= [g\,(\phi_1)\,/\,s_1, g\,(\phi_2)\,/\,s_2]\,(s_1 \vee s_2) \\
g\,(\phi_1 \vee \phi_2) &= [g\,(\phi_1)\,/\,s_1, g\,(\phi_2)\,/\,s_2]\,(s_1 \wedge s_2)
\end{aligned}$$

In our proof we need the following lemma:

Lemma 1. *For any counting function f and formulae* ϕ *and* ψ*,* $f\,([\phi\,/\,p]\;\psi)$ *is a linear expression in* $f\,(\phi)$*.*

Proof. Proof with induction on ψ:
Since f is a counting function, there exists constants c_i, a, b, $a_\square$, $b_{1\square}$, and $b_{2\square}$ such that

$$\begin{aligned}
f\,(p_i) &= c_i, \\
f\,(\neg\phi) &= a + b \cdot f\,(\phi), \\
f\,(\phi_1 \,\square\, \phi_2) &= a_\square + b_{1\square} \cdot f\,(\phi_1) + b_{2\square} \cdot f\,(\phi_2)
\end{aligned}$$

For atomic formulae ψ, $f\,([\phi\,/\,p]\;\psi)$ is either $f\,(p_i)$ (a constant) or $f\,(\phi)$ and hence linear in $f\,(\phi)$.
For the inductive cases we assume that $f\,([\phi\,/\,p]\;\psi_1) = c_1 + d_1 \cdot f\,(\phi)$ and $f\,([\phi\,/\,p]\;\psi_2) = c_2 + d_2 \cdot f\,(\phi)$.
Then:

$$\begin{aligned}
& f\,([\phi\,/\,p]\,(\neg\psi_1)) \\
=\; & f\,(\neg[\phi\,/\,p]\;\psi_1) \\
=\; & a + b \cdot f\,([\phi\,/\,p]\;\psi_1) \\
=\; & a + b \cdot (c_1 + d_1 \cdot f\,(\phi)) \\
=\; & a + b \cdot c_1 + b \cdot d_1 \cdot f\,(\phi)
\end{aligned}$$

And:

$$\begin{aligned}
& f\,([\phi\,/\,p]\,(\psi_1 \,\square\, \psi_2)) \\
=\; & f\,([\phi\,/\,p]\;\psi_1 \,\square\, [\phi\,/\,p]\;\psi_2) \\
=\; & a_\square + b_{1\square} \cdot f\,([\phi\,/\,p]\;\psi_1) + b_{2\square} \cdot f\,([\phi\,/\,p]\;\psi_2)
\end{aligned}$$

$$= a_\Box + b_{1\Box} \cdot (c_1 + d_1 \cdot f\ (\phi)) + b_{2\Box} \cdot (c_2 + d_2 \cdot f\ (\phi))$$
$$= a_\Box + b_{1\Box} \cdot c_1 + b_{2\Box} \cdot c_2 + (b_{1\Box} \cdot d_1 + b_{2\Box} \cdot d_2) \cdot f\ (\phi)$$

□

In the same way we can prove that for any counting function f and formulae ϕ_1, ϕ_2 and ψ, $f\ ([\phi_1 / s_1, \phi_2 / s_2]\ \psi)$ is a linear expression in $f\ (\phi_1)$ and $f\ (\phi_2)$. We omit the proof. Using Lemma 1 we can prove the next lemma:

Lemma 2. *For any counting function f and acceptable inductively defined function g, after applying g*

- *$f\ (g\ (p_i))$ is a constant for atomic formulae p_i*
- *$f\ (g\ (\neg\phi))$ is a linear expression in $f\ (g\ (\phi))$*
- *$f\ (g\ (\phi_1 \Box \phi_2))$ is a linear expression in $f\ (g\ (\phi_1))$ and $f\ (g\ (\phi_2))$*

Proof. For atomic formulae, $g\ (p_i)$ is a propositional formula, and hence, $f\ (g\ (p_i))$ is a constant.
Since g is acceptable, there exists a formula ψ and variable s such that $f\ (g\ (\neg\phi)) = f\ ([g\ (\phi) / s]\ \psi)$, which is linear in $f\ (g\ (\phi))$ according to Lemma 1.
In the same way: there exists a formula $\psi_\Box$ and variables s_1 and s_2 such that $f\ (g\ (\phi_1 \Box \phi_2)) = f\ ([g\ (\phi_1) / s_1, g\ (\phi_2) / s_2]\ \psi_\Box)$, which is linear in $f\ (g\ (\phi_1))$ and $f\ (g\ (\phi_2))$ by the generalization of Lemma 1. □

Theorem 1. *The strategy used by LogInd can generate an inductive proof for any correct statement of the form $P_1\ (\phi)$ comp $P_2\ (\phi)$, where comp is a comparator ($=, <, \leqslant, >, \geqslant$), $P_2\ (\phi)$ is a linear combination of expressions $f_i\ (g_i\ (\phi))$, and $P_1\ (\phi)$ is a single expression $f\ (g\ (\phi))$, for which f and f_i are counting functions, and g and g_i are inductively defined functions from L to L.*

Proof. The strategy starts with the application of the inductively defined functions. We first consider the base case. After the application of inductively defined functions, the left-hand side of the equation is a constant and the right-hand side a linear combination of constants, which is rewritten into a single constant using arithmetic, by Lemma 2. A number comparison suffices to check if the base case holds. In the inductive cases, the application of the inductively defined function in the left-hand side results in a linear expression in $f\ (g\ (\phi))$ (case negation) or $f\ (g\ (\phi_1))$ and $f\ (g\ (\phi_2))$ (case binary connective). The right-hand side is a linear combination of linear expressions in $f_i\ (g_i\ (\phi))$ or in $f_i\ (g_i\ (\phi_1))$ and $f_i\ (g_i\ (\phi_2))$. The next step in the algorithm is the application of the induction hypothesis. Replacing occurrences of $f\ (g\ (\phi))$ or $f\ (g\ (\phi_1))$ and $f\ (g\ (\phi_2))$ by the right-hand side of the induction hypothesis results in another linear combination of $f_i\ (g_i\ (\phi))$ or $f_i\ (g_i\ (\phi_1))$ and $f_i\ (g_i\ (\phi_2))$. Normalizing both left-hand side and right-hand side now suffices to prove the inductive cases. □